# An Interactive Environment for Data Partitioning and Distribution

Vasanth Balasundaram* Geoffrey Fox* Ken Kennedy† Ulrich Kremer†

**Abstract**

An approach to distributed memory parallel programming that has recently become popular is one where the programmer explicitly specifies the data decomposition using language extensions, and a compiler generates all the communication. While this frees the programmer from the tedium of thinking about message-passing, no assistance is provided in determining the data decomposition scheme that gives the best performance on the target machine. In this paper, we propose an interactive software tool that provides assistance for this very task. The proposed tool also computes performance estimates for any chosen data partitioning scheme, allowing the programmer to experiment with several different strategies without ever running the program on the machine.

Although distributed memory parallel computers are among the most cost-effective machines available, most scientists find them difficult to program. The reason is that traditional programming languages support shared name spaces and, as a result, most programmers feel more comfortable working with a shared memory programming model. To this end, a number of researchers [6, 16, 4, 19, 15, 12, 5] have proposed using a traditional sequential or shared-memory language extended with annotations specifying how the data is to be distributed. This approach is inspired by the observation that the most demanding intellectual step in programming for distributed memory is the data layout — the rest is straightforward but tedious and error prone work. Given a program and annotations of this sort, a compiler can mechanically generate the node program for a distributed-memory machine. This strategy is illustrated by steps II and III in Figure 1.

A problem with this approach is that it provides no useful feedback to the user concerning the effectiveness of the decisions about data layout, other than running time. What is needed is a tool that will assist the user in selecting the partitioning and distribution of each array in the program. In this paper we describe the design of just such a tool (labeled step I in Figure 1). The key ideas behind this tool are (1) reasonably simple static models can be used to estimate the performance of a program under various data distributions and (2) if we restrict ourselves to fairly simple partitionings, then for a program segment such as a loop, there are only a small number of such partitionings suitable for each array and hence these distributions can be exhaustively examined by the user, and (3) the data partitioning for the entire program can be done by successively partitioning the data for smaller program segments, and repartitioning when necessary between the program segments.

We begin in Section 1 with an example that illustrates the difficulty of choosing a good data partitioning strategy, and motivate the need to provide some assistance in this task. Section 2 describes the overall design of the proposed tool. Section 3 briefly explains how data dependences in the program influence the choice of a data partitioning strategy. Once the programmer chooses a particular data partitioning scheme, the tool determines the communication required, and returns a cost estimate based on a performance model. This analysis, which is done by the performance estimation module, is described in Section 4, and is the primary focus of this paper.

## 1 Finding a good data partitioning: an example

The overall performance of a distributed memory program is affected by factors such as the program's data size, target machine specific parameters, and the chosen partitioning scheme. In this section, we will briefly examine the relationship between these aspects, illustrating the subtle complexities that must be taken into account by the user in order to find the best data par-

*Caltech Concurrent Computation Program, Mail Stop 206-49, Caltech, Pasadena, CA 91125.

†Dept. of Computer Science, Rice University, Houston, TX 77251. Please address all correspondence regarding this paper to the presenting author, Ulrich Kremer (kremer@rice.edu).



0-8186-2113-3/90/0000/1160$01.00 © 1990 IEEE

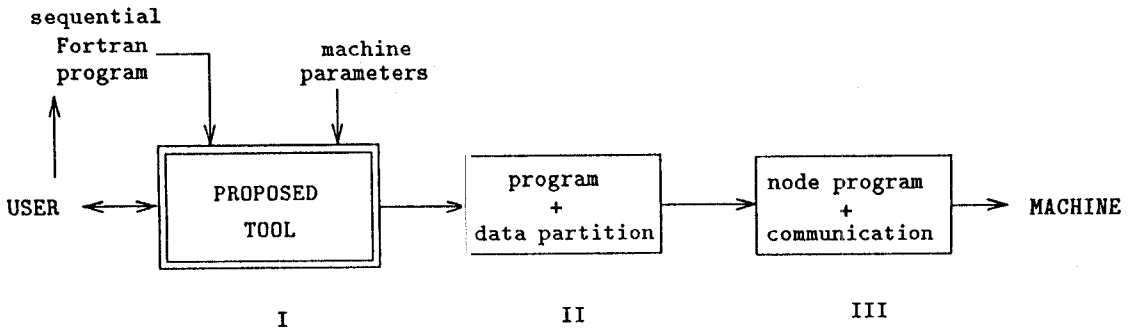


FIGURE 1: The program development process.

titioning. In the following program segment, $\mathcal{F}$ and $\mathcal{F}'$ represent functions with four and ten double precision floating point operations, respectively. The program segment was executed on 64 processors of an NCUBE, with array sizes ranging from N = 64 to N = 320. The resulting execution and communication times for column and block partitioning schemes are shown graphically in Figure 2. The communication time was measured by removing all computation in the loops.

```
subroutine example (A, B, N)
    double precision A(N, N), B(N, N)

    do k=1, cycles
        do j=1,N
            do i=2,N-1
                A(i, j) = F ( B(i-1, j), B(i+1, j) )
            enddo
        enddo
        do j=2,N-1
            do i=2,N-1
                B(i,j) = F' ( A(i-1, j), A(i+1, j), A(i, j),
                    A(i, j-1), A(i, j+1) )
            enddo
        enddo
    enddo
end
```

When employing a column partitioning scheme for arrays A and B, communication is only necessary after the first j loop. Each processor has to exchange boundary values with its left and right neighbor. In a block partitioning scheme each processor has to communicate with its four neighbors after the first loop and with its neighbors in the north and south after the second loop. For small message lengths the communication cost is dominated by the fixed startup time whereas the transmission cost begins to dominate as the messages get longer (i.e., more data is exchanged at each communication step). This explains why communication cost for the column partition is greater than for the block partition for array sizes larger than 128 x 128. It is clear from the graph that column partitioning is preferable when the array sizes are less than 128 x 128, and block partitioning is preferable for larger sizes.

The steps in the execution time graphs are caused mainly by load imbalance effects. For example, the step between N = 128 and N = 129 for the column partition is due to the fact that for size 129 one subdomain has an extra column, so that the processor assigned to that subdomain is still busy after all the others have finished, causing load imbalance in the system. Similar behavior can be observed for the block partition but here the steps occur at smaller increments of the array size N. The steps in the communication time graphs are due to the fact that the packet size on the NCUBE is 1Kbyte, so that messages that are even a few bytes longer need an extra packet to be transmitted.

The above example indicates that several factors contribute to the observed performance of a chosen partitioning scheme, making it difficult for a human to predict this behavior. What we need is an interactive tool that allows the user to gauge the behavior of a partitioning scheme without having to either rewrite the program or run the program on the target machine.

## 2 Overview of the tool

When using the tool we envision, the programmer will select a program segment for analysis and the system will provide assistance on which partitionings to choose for that program segment, for various problem sizes. The assistance provided by the tool has two flavors. Data dependence information supports the user in determining a set of reasonable partitionings, and performance estimates help the user in choosing an efficient data partitioning strategy.

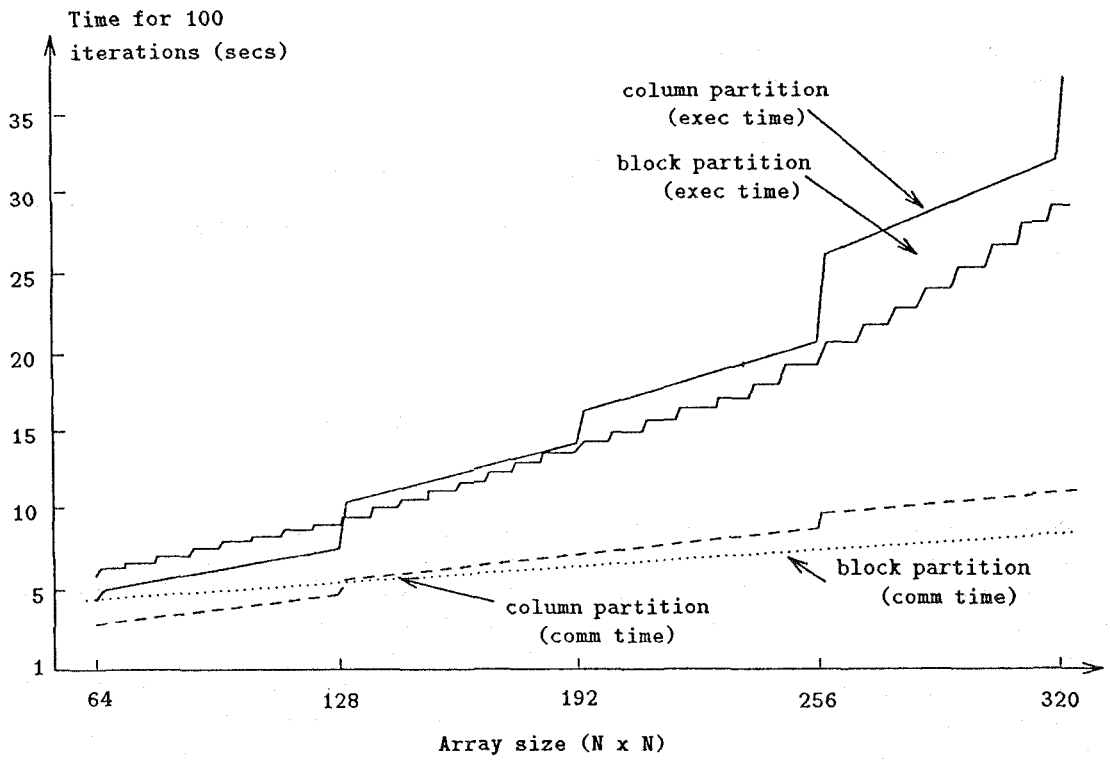


FIGURE 2: Timing results on an NCUBE, using 64 processors.

The user first chooses a data partitioning scheme, based on an analysis of the data dependences in the program. An important component of the system is the performance estimation module, which is subsequently used to select the best partitionings and distributions from among those examined. In the present version, the do loop is the only kind of program segment that can be selected. For simplicity, the set of possible partitions of an array is restricted to regular patterns such as by row, by column or by block for a two dimensional array and their higher dimensional analogs for arrays of larger dimensions. All the partitions are assumed to be homogeneous, i.e., approximately the same size and shape. This permits the examination of all reasonable partitionings of the data in an acceptable amount of time. Based on the performance cost estimates returned by the tool, the user can modify the data partitioning. The above process is then repeated until the user is satisfied with the predicted performance of the chosen data partitioning scheme.

Initially, we propose to use a purely static model for performance prediction using architectural parameters of the target machine and the size of data sets as the principal inputs. Since communication cost is an important indicator of performance on a distributed memory parallel computer, in the interest of clarity, we will concentrate our discussion on this measure.

As depicted in Figure 1, the proposed tool will generate an annotated program as its output, which in turn is input to a compiler as discussed earlier. In order to be able to predict the communication costs for a program segment, the tool must have knowledge about the basic compilation strategy. In this discussion, we assume that the compiler does not perform any program restructuring transformations. The compiler may, however, do simple communication optimizations, such as merging smaller messages into longer ones. We are currently working on the possibility of having the internal analysis mimic restructuring optimizations that the compiler would perform, given the current program as an input.

We assume that each processor executes only those program statement instances that define a value of a data item that has been mapped on to that processor by the partitioning and distribution specifications. Data items that are mapped on to a processor are said to be *owned* by that processor. Execution of such a statement may require non-local data, i.e., data that is owned by another processor. Such non-local data items must be obtained by communication.

The tool permits the user to generalize from local partitionings to layouts for an entire program in easy steps, using repartitioning and redistribution when-

ever it leads to better performance overall. The principal value of the environment for partitioning and distribution is that it supports an exploratory programming style in which the user can experiment with different data partitioning strategies and estimate the effect of each strategy for different input data sizes or different target machines without having to change the program or run the program each time.

In the following section, we describe the use of the performance estimation module, using a simple program segment as an example.

## 3 Dependence-based data partitioning

Given a sequential Fortran program, and a selected program segment (which in the preliminary version can only be a loop nest), the tool provides assistance in deriving a set of reasonable data partitions for the arrays accessed in that segment. The assistance is given in the form of data dependence information for variables accessed within the selected segment. When partitioning data, we must ensure that the parallel computations done by all the processors on their local partitions preserve the data dependence relations in the sequential program segment. If the computations done by the processors on the distributed data satisfy all the data dependences, the results of the computation will be the same as that produced by a sequential execution of the original program segment. There are two ways to achieve this: (1) by "internalizing" data dependences within each partition, so that all values required by computations local to a processor are available in its local data subdomain, or (2) by inserting appropriate communication to get the non-local data.

Let us consider a sample program segment, and see how data dependence information can be used to help derive reasonable data partitionings for the arrays accessed in the segment.

**P1. Example program segment.**

```
do j = 2, n
   do i = 2, n
      A(i, j) = F( A(i−1, j) )
      B(i, j) = F'( A(i, j), B(i, j−1), B(i, j) )
   enddo
enddo
```

$\mathcal{F}$ and $\mathcal{F}'$ are functions whose exact nature is irrelevant to this discussion. When the programmer selects the "do i" loop, the tool indicates that there is one data dependence that is carried by the i loop: the dependence of A(i, j) on A(i−1, j). This dependence indicates that the computation of an element of A cannot be started until the element immediately above it in the previous row has been computed. The programmer then selects the outer "do j" loop to get the data dependences that are carried by the j loop. There is one such dependence, that of B(i, j) on B(i, j−1). This dependence indicates that the computation of an element of B cannot be started until the computation of the element immediately to the left of it in the previous column has been computed. Figure 3(a) illustrates the pattern of data dependences for the above program segment.

The pattern of data dependences between references to elements of an array gives the programmer clues about how to partition the array. It is usually a good strategy to partition an array in a manner that internalizes all data dependences within each partition, so that there is no need to move data between the different partitions that are stored on different processors. This avoids expensive communication via messages. For example, the data dependence of A(i, j) on A(i−1, j) can be satisfied by partitioning A in a column-wise manner, so that the dependences are "internalized" within each partition. The data dependence of B(i, j) on B(i, j−1) can be satisfied by partitioning B row-wise, since this would internalize the dependences within each partition.

It is not enough to examine only the dependences that arise due to references to the same array. In some cases, the data flow in the program implicitly couples two different arrays together, so that the partitioning of one affects the partitioning of the other. In our example, each point B(i, j) also requires the value A(i, j). We treat this as a special data dependence called a *value* dependence (read "B is value dependent on A"), to distinguish it from the traditional data dependence that is defined only between references to the same array. This value dependence must also be satisfied either by internalization or by communication. Internalization of the value dependence is possible only by partitioning B in the same manner as A, so that each B(i, j) and the A(i, j) value required by it are in the same partition.

Based on the pattern of data dependences in the program segment, the following are a possible list of partitioning choices that can be derived:

(a) Partition A by column and B by column. This satisfies the dependences within A and the value dependences of B on A by internalization but communication is required to satisfy the data dependences within B (Figure 3(a)). An analogous case is to partition both A and B by row. This would require communication to satisfy dependences within A.

(b) Partition A by column and B by row. Dependences within B are now satisfied by internalization, but communication is needed to satisfy the value dependence of B on A (Figure 3(b)).

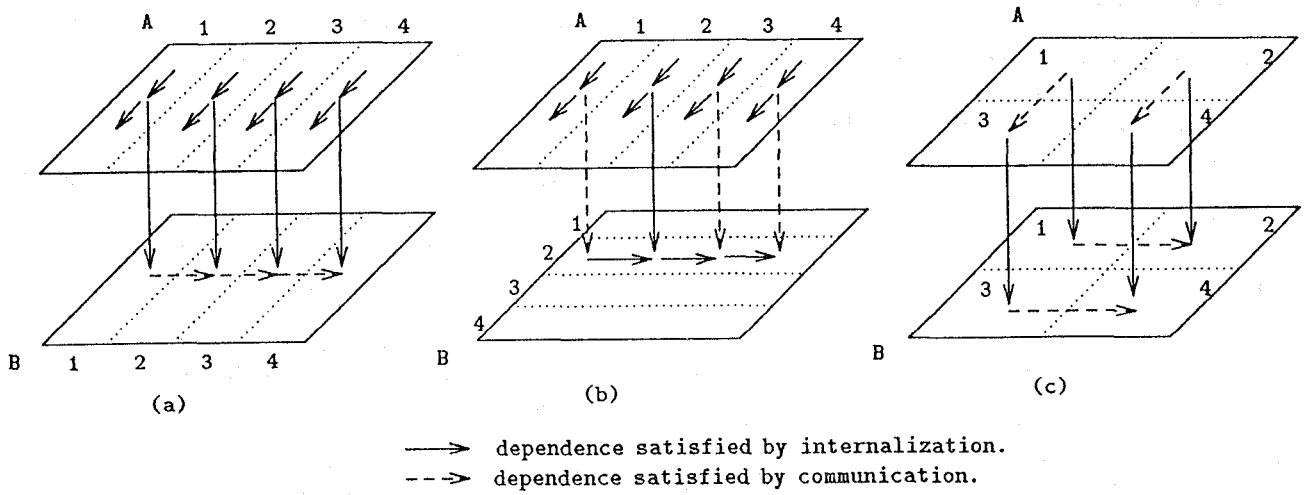


FIGURE 3: Data dependences satisfied by internalization and communication for the partitioning schemes (a) A by column, B by column (b) A by column, B by row and (c) A by block, B by block. Dotted lines represent partition boundaries and numbers indicate virtual processor IDs (the figures are shown for p = 4 virtual processors). For clarity, only a few of the dependences are shown.

(c) Partition both A and B as 2 dimensional blocks. This would result in communication to satisfy dependences within both A and B, while the value dependence of B on A is satisfied by internalization (Figure 3(c)).

The partitioning of A by row and B by column was not considered among the possible choices because in this scheme, none of the dependences are internalized, thus requiring greater communication compared to (a), (b) or (c). Communication overhead is a major cause of performance degradation on most machines, so a reasonable first choice would be the partitioning scheme that requires the least communication. This can be determined either by analyzing the number of dependences that are cut by the partitioning (indicating the need for communication), or more accurately using the performance estimation module that is described in the next section.

# 4 The performance estimation module

In a first step, the performance estimation module computes an internal data mapping of the specified data partitioning. This information is subsequently used to determine the necessary communications for the given program segment under the specified partitioning. The results of the communication analysis are passed to the static performance estimator which determines a relative cost estimate for the communication time and overall execution time of the program segment.

## 4.1 Mapping data to processors

For the selected program segment, the programmer picks one of the choices (a)-(c), and specifies the data partitioning and distribution via an interface provided by the tool. The tool responds by creating an internal data mapping that specifies the mapping of the data to a set of *virtual processors*. The number of virtual processors is equal to the number of partitions indicated by the data partitioning. The mapping of the virtual processors onto the physical processors is assumed to be done by the run-time system, and this mapping is unspecified in the software layer. Henceforth, we will use the term "processor" synonymously with "virtual processor".

### 4.1.1 Distribution of arrays

Let us continue with our example program segment, and see how the internal mapping is constructed for partitioning (b), i.e., A partitioned by column and B by row. The data mappings for the other two cases can be constructed in a similar manner. Let A and B be of size n × n and the number of (virtual) processors be p. For simplicity we assume that p divides n. The following two data mappings are computed:

- A(1:n, 1:n) partitioned by column: Create a virtual array A$(1:p), where A$(k) represents the kth column partition of A, i.e., A$ consists of the elements A(1:n, (n/p)(k−1)+1:(n/p)k). The virtual array is only an internal entity, used within the tool to maintain the mapping of data to (virtual) processors. It does not have any physical storage on the machine. The partition of A represented by A$(k) is assumed to be mapped onto the kth processor by default.

- B(1:n, 1:n) partitioned by row: Create a virtual array B$(1:p), where B$(k) represents the ith row partition of B, i.e., B$ consists of the elements B((n/p)(k−1)+1:(n/p)k, 1:n). B$(k) is assigned to the kth processor by default.

The internal data mapping is used to solve the following two problems:

(I) Given a processor q, what part of A is local to it? This is given by the section of A that belongs to the partition A$(q).

(II) Given a section A(x1:x2, y1:y2), what processors contain elements of this section? This is given by the set of processors $\{q \mid A\$(q) \cap A(x1:x2, y1:y2) \neq \phi\}$.

The values n and p are assumed to be known statically. The section information for each partition is actually kept in a structure called the Data Access Descriptor, that provides a compact representation for array sections of different shapes, and allows fast intersections. Details about the Data Access Descriptor representation are beyond the scope of this paper, and can be found elsewhere [2]. To simplify the treatment in this paper, we will simply treat a Data Access Descriptor as being a set of value ranges, one for each dimension of the array. Given an array reference and a loop that encloses it, we can compute the range of values that each dimension of the array can take for all iterations of the loop. This describes the (rectangular) section of the array that is accessed within the loop.

A useful property of Data Access Descriptors is the ability to incrementally "translate" an accessed section computed with respect to a particular loop, to the section accessed with respect to an enclosing loop. For example, consider a reference to a 2 dimensional array within a doubly nested loop. The section of the array accessed within each iteration of the innermost loop is a single element. The same reference when evaluated with respect to the entire inner loop (i.e., all iterations of the inner loop) may access a column of the array. If we evaluated the reference with respect to the outer loop (i.e., all iterations of the outer loop), we may notice that the reference results in an access of the entire array, in a column-wise manner. This method of converting array sections in terms of enclosing loops is called *translation*, and is denoted by the symbol "⇑".

The tool uses (I) to determine which processors should do what computations. As mentioned in a previous section each processor executes only those program statement instances which assign into a data item that the processor owns. To perform these assignments a processor might need values of data items that it does not own. The inverse mapping (II) is used to determine the set of processors that own the desired values. These processors must send the data item they own to the processor that will execute the statement instance.

#### 4.1.2 Distribution of scalars: replication

The data mapping scheme described above works only for arrays. Scalar variables are assumed to be replicated, i.e., every processor stores a copy of the scalar variable in its local memory and therefore owns the variable. By the rule stated earlier, this implies that any statement that computes the value of a scalar is executed by all the processors.

### 4.2 Communication analysis

The communication analysis algorithm takes the internal data mappings, the dependence graph, and the loop nesting structure of the specified program segment as its input. For each processor the algorithm determines information about all communications the processor is involved in. We will now illustrate the communication analysis algorithm using the example program segments P1, P2, and P3, where P2 is derived from P1, and P3 from P2, respectively, by a transformation called *loop distribution*.

#### 4.2.1 Performance improvement transformations

Substantial performance improvement can be achieved by performing various code transformations on the program segment. For example, the *loop-distribution* transformation [18, 1] often helps reduce the overhead of communication. Loop-distribution splits a loop into a set of smaller loops, each containing a part of the body of the original loop. Sometimes, this allows communication to be done between the resulting loops, which may be more efficient than doing the communication within the original loop.

Consider the program segment P1. If A is partitioned by column and B by row, communication will be required within the inner loop to satisfy the value dependence of B on A. Each message communicates a single element of A. For small message sizes and large number of messages, the fraction of communication

time taken up by message startup overhead is usually quite large. Thus, program P1 will most likely give poor performance because it involves the communication of a large number of small messages.

However, if we loop-distributed the inner `do i` loop over the two statements, the communication of A from the first `do i` loop to the second `do i` loop can be done between the two new inner loops. This allows each processor to finish computing its entire column partition of A in the first `do i` loop, and then send its part of A to the appropriate processors as larger messages, before starting computation of a partition of B in the second `do i` loop. This communication is done only once for each iteration of the outer `do j` loop, i.e., a total of O(n) communication steps. In comparison, program P1 requires communication within the inner loop, which gives a total of $O(n^2)$ communication steps:

**P2. After loop-distribution of i loop.**

```
do j = 2, n
    do i = 2, n
        A(i, j) = F( A(i−1, j) )
    enddo
    do i = 2, n
        B(i, j) = F'( A(i, j), B(i, j−1), B(i, j) )
    enddo
enddo
```

The reduction in the number of communication steps also results in greater parallelism, since the two inner `do i` loops can be executed in parallel by all processors, without any communication. This effect is much more dramatic if we apply loop-distribution once more, this time on the outer `do j` loop:

**P3. After loop-distribution of j loop.**

```
do j = 2, n
    do i = 2, n
        A(i, j) = F( A(i−1, j) )
    enddo
enddo
do j = 2, n
    do i = 2, n
        B(i, j) = F'( A(i, j), B(i, j−1), B(i, j) )
    enddo
enddo
```

For the same partitioning scheme (i.e., A by column and B by row), we now need only O(1) communication steps, which occur between the two outer `do j` loops. The computation of A in the first loop can be done in parallel by all processors, since all dependences within A are internalized in the partitions. After that, the required communication is performed to satisfy the value dependence of B on A. Then the computation of B can proceed in parallel, because all dependences within B are internalized in the partitions. The absence of any communication within the loops considerably improves efficiency.

Currently, the tool provides a menu of several program transformations, and the programmer can choose which one to apply. When a particular transformation is chosen by the programmer, the tool responds by automatically performing the transformation on the program segment, and updating all internal information automatically.

### 4.2.2 Communication analysis algorithm

For the sake of illustration, let the size of A and B be $8 \times 8$ (i.e., $n = 8$), and let the number of (virtual) processors be $p = 4$. The following is a possible sequence of actions that the programmer could do using the tool.

After examining the data dependences within the program segment as reported by the tool, let us assume that the programmer decides to partition A by column and B by row. The tool computes the internal mapping:

A\$(1) = A(1:8, 1:2) and B\$(1) = B(1:2, 1:8).
A\$(2) = A(1:8, 3:4) and B\$(2) = B(3:4, 1:8).
A\$(3) = A(1:8, 5:6) and B\$(3) = B(5:6, 1:8).
A\$(4) = A(1:8, 7:8) and B\$(4) = B(7:8, 1:8).

To determine the communication necessary, the tool uses Algorithm COMM, shown in Figure 4. For simple partitioning schemes as found in many applications, the communication computed by algorithm COMM can be parameterized by processor number, i.e., evaluated once for an arbitrary processor. In addition, we are also investigating other methods to speed up the algorithm.

Consider program P1 for example. According to algorithm COMM, when the kth processor executes the first statement, the required communication is given by

$$\{(q, \lambda) \mid \lambda = A\$(q) \cap A(i-1, j) \neq \phi\}$$

where the range of i and j are determined by the section of the LHS owned by processor k, in this case i = 2:8 and j = 2(k−1)+1:2k (since A is partitioned column-wise). But the partitioning of A ensures that $\forall$ k, the data A(∗, 2(k−1)+1:2k) is always local to k. The set of $(q,\lambda)$ pairs will therefore be an empty set for any k. Thus, the execution of the first statement with A partitioned by column requires no communication.

When the kth processor executes the second statement, the communication as computed by Algorithm COMM is given by

$$\{(q,\lambda) \mid \lambda = A\$(q) \cap A(i, j) \neq \phi\}$$
$$\cup \quad \{(q,\lambda) \mid \lambda = B\$(q) \cap B(i, j-1) \neq \phi\}$$
$$\cup \quad \{(q,\lambda) \mid \lambda = B\$(q) \cap B(i, j) \neq \phi\}.$$

The ranges of i and j are determined by the section of the LHS that is owned by processor k, in this case i

**Algorithm COMM**

*Input:* The data mapping specified by the chosen partitioning scheme, the dependence graph, and the selected loop.
*Output:* A set of pairs $(q,\lambda)$, indicating that processor q must send the section $\lambda$ of data that it owns, to processor k, and the level at which the communication occurs.

```
for each processor k do
    for each statement do
        Let def(X) be the section of the LHS array X that is owned by the kth processor;
        for each RHS array reference Y do
            Let use(Y) be the section of the RHS array Y that is needed to compute each element of def(X);
            We need to determine the communication required, if any, to get use(Y) from all processors q ≠ k;
            Define commlevel of a dependence to be:
                { level of the dependence,                                  if it is loop-carried
                { common nesting level of src and sink of dependence,       if it is loop-independent
            Let lmax = max(commlevels of all dependences with Y as sink reference);
            Let ⇑use(Y) be the section use(Y) "translated" to the level lmax;
            Then, the set of all (q,λ) pairs is given by {(q,λ), q ≠ k | λ = Y$(q) ∩ ⇑use(Y) ≠ φ},
              with the communications occurring at level lmax;
        endfor
    endfor
endfor
```

FIGURE 4: Algorithm to determine the communication induced by the data partitioning scheme.

= 2(k−1)+1:2k and j = 2:8 (since B is partitioned row-wise). The second and third terms will be $\phi$, because the row partitioning of B ensures that $\forall$ k, the data B(2(k−1)+1:2k, *) is always local to k. The first term can be a non-empty set, because processor k owns a column of A (i.e., j in the range 2(k−1)+1:2k)), while the range of j in the first term is 2:8. Thus, communication may be required to get the non-local element of A before the kth processor can proceed with the computation of its B(i, j). The dependence from the definition of A(i,j) to its use is loop-independent. Algorithm COMM therefore computes `commlevel`, the common nesting level of the source and sink of the dependence, to be the level of the inner i loop. The section A(i,j) translated to the level of the inner i loop is simply the single element A(i,j). Thus each message communicates this single element, and the communication occurs within the inner i loop.

The execution of program P1 results in a large number of messages because each message only communicates a single element of A, and the communication occurs within the inner loop. Message startup and transmission costs are specified by the target machine parameters, and the average cost of each message is determined from the performance model. The tool computes the communication cost by multiplying the number of messages by the average cost of sending a single element message. This cost estimate is returned to the programmer.

Now consider the program P2, with the same partitioning scheme for A and B. When the kth processor executes the first statement, the required communication as determined by Algorithm COMM is given by

$$\{(q,\lambda) \mid \lambda = A\$(q) \cap A(1{:}7, j) \neq \phi\}$$

where the range of j is determined by the section of the LHS owned by processor k, in this case j = 2(k−1)+1:2k (since A is partitioned column-wise). Note that in this case, ⇑A(i−1,j) = A(1:7,j). This is because `commlevel` is now the level of the outer j loop, so that the section A(i−1,j) must be translated to the level of the j loop. In other words, the reference to A(i−1,j) in the first statement results in an access of the first 7 elements of the jth column of A, during each iteration of the j loop. Since A is partitioned column-wise, this section will always be available locally in each processor, so that the above set is empty and no communication is required.

When processor k executes the second statement, the communication required is given by

$$\{(q,\lambda) \mid \lambda = A\$(q) \cap A(2(k-1)+1{:}2k, j) \neq \phi\}$$
$$\cup \quad \{(q,\lambda) \mid \lambda = B\$(q) \cap B(2(k-1)+1{:}2k, j-1) \neq \phi\}$$
$$\cup \quad \{(q,\lambda) \mid \lambda = B\$(q) \cap B(2(k-1)+1{:}2k, j) \neq \phi\}$$

The second and third terms of will be empty sets since the required part of B is local to each k (because B is partitioned row-wise). The first term will be non-empty, because each processor owns A(*, 2(k−1)+1:2k), and the range of j in the first term is outside the range 2(k−1)+1:2k. The data required by

processor k from processor q will therefore be a strip A(2(k−1)+1:2k, j), from each q ≠ k.

This data can be communicated between the two inner `do i` loops. Each message will communicate a 2×1 size strip of A. Fewer exchanges will be required compared to program P1, because each exchange now communicates a strip of A, and the communication occurs outside the inner loop. Once again, the performance model and target machine parameters are used by the tool to estimate the total communication cost, and this cost is returned to the programmer.

For most target machines, the communication cost in program P2 will be considerably less than in program P1, because of larger message size and fewer messages.

Next, let us consider program P3. Assuming that the same partitioning scheme is used for A and B, the execution of the first loop by the kth processor will require communication given by

$$\{(q,\lambda) \mid \lambda = A\$(q) \cap A(1{:}7,\ 2(k-1)+1{:}2k)\}.$$

But this is an empty set because of the column partitioning of A. Here ⇑A(i−1,j) = A(1:7, 2(k−1)+1:2k), because `commlevel` for this case is the level of the subroutine that contains the two loops. The section is therefore translated to this level, by substituting the appropriate bounds for i and j. The translated section indicates that the reference A(i−1,j) in the first statement results in an access of the section A(1:7, 2(k−1)+1:2k) during all iterations of the outer j loop that are executed by processor k.

When the kth virtual processor executes the second loop, the required communication is

$$\{(q,\lambda) \mid \lambda = A\$(q) \cap A(2(k-1)+1{:}2k,\ 2{:}8) \neq \phi\}$$
$$\cup\ \{(q,\lambda) \mid \lambda = B\$(q) \cap B(2(k-1)+1{:}2k,\ 1{:}7) \neq \phi\}$$
$$\cup\ \{(q,\lambda) \mid \lambda = B\$(q) \cap B(2(k-1)+1{:}2k,\ 2{:}8) \neq \phi\}.$$

The second and third terms will be empty sets because of the row partitioning of B. The first term will be non-empty, and the data required by processor k from processor q will be the block A(2(k−1)+1:2k, 2(q−1)+1:2q), for each q ≠ k. This block can be communicated between the two `do j` loops.

This communication can be done between the two loops, allowing computation within each of the two loops to proceed in parallel. The number of messages is the fewest for this case because a 2×2 block of A is communicated during each exchange. Program P3 is thus likely to give superior performance compared to P1 or P2, on most machines.

### 4.3 Static performance estimator

Given the results of the communication analysis in a program segment, the performance estimator can be used to predict the performance of that program segment on the target machine. It uses a simple static model of performance that is based on (1) target machine parameters such as the number of processors, the message startup and transmission costs and the average times to perform different floating point operations, (2) the size of the input data set, and (3) the data partitioning scheme.

Many aspects of our performance model have been borrowed from published studies [7, 10, 17, 5]. The static performance model is meant primarily to help the programmer discriminate between different data partitioning schemes, rather than give an accurate estimate of running time. The exact details of the performance model are beyond the scope of this paper.

We ran programs P1, P2 and P3 with A partitioned by column and B by row, on 16 processors of the NCUBE at Caltech. The functions $\mathcal{F}$ and $\mathcal{F}'$ consisted of one and two double precision floating point operations, respectively. The results of the experiment are shown in Figure 5. The graphs clearly illustrate the performance improvement that occurs due to reduction in number of messages and increase in length of each message. We were also able to predict the overall execution and communication time of our first program example as shown in Figure 2 with high accuracy.

## 5 Related work

Several researchers are developing compilers that take a program with annotations for specifying data distribution, and generate the necessary communication primitives. The Superb project at Bonn University [19, 9], the Kali project at Purdue University and NASA-ICASE [6, 12], Callahan and Kennedy's work at Rice University [4] and the work of Rogers and Pingali [15] are examples of this approach. The Crystal project at Yale University [5] is also based on the same idea, but targeted primarily for the functional language Crystal. The work of Ramanujan and Sadayappan at Ohio State University [14] attempts to automatically derive data partitionings for a restricted class of programs. Other approaches to automatic data partitioning are discussed in Knobe, Lukas and Steele [11] for SIMD architectures like the Conncetion Machine, and Li and Chen [13] in the context of the Crystal project.

Our approach is to provide the programmer with the necessary tools to experiment with several data partitioning strategies, until he can converge on the one that is likely to give him a satisfactory performance. The tool provides feedback information about performance estimates each time a partitioning is done by the programmer.

## 6 Future work

Our emphasis is to try to recognize *collective* communication patterns rather than generating sequences of

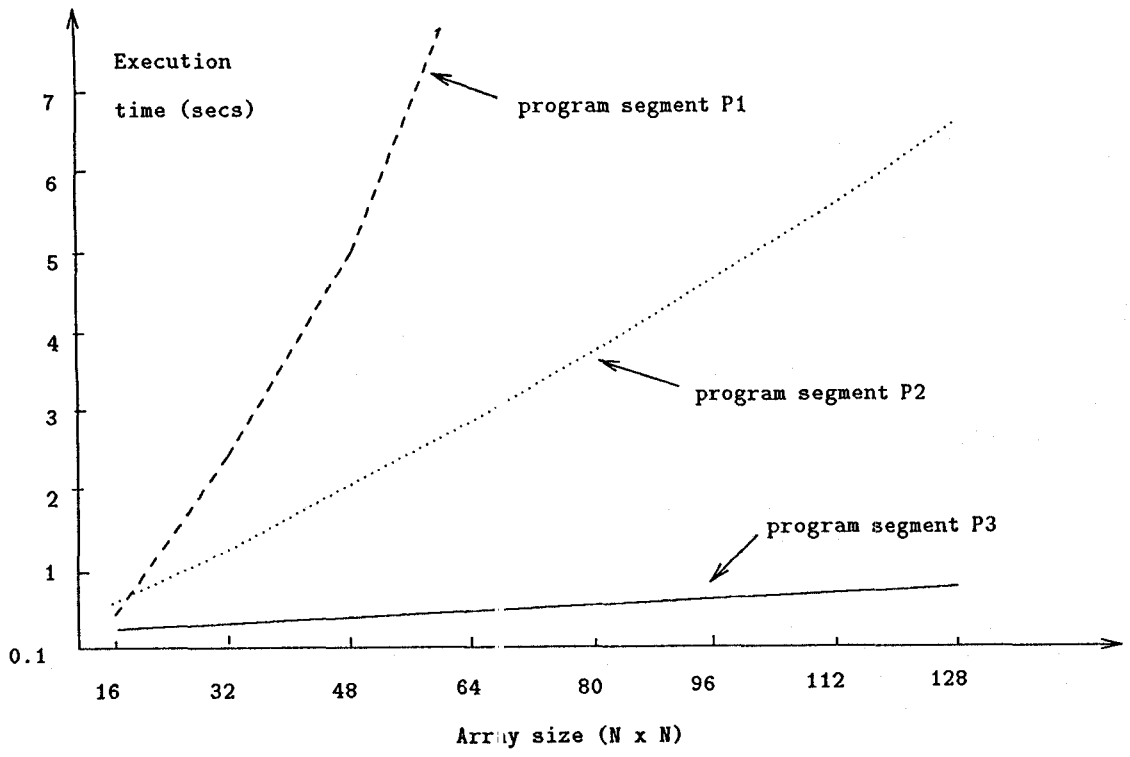


FIGURE 5: Timing results for programs P1, P2 and P3 on the NCUBE, using 16 processors.

sends and receives. Algorithm COMM (Figure 4) determines this in a very natural way. This is especially important for *loosely synchronous problems* which represent a large class of scientific computations [7] Such problems can be characterized by computation intensive regions that have substantial parallelism with communication required between the regions. Several communication utilities have been developed that provide optimal message passing communication for such problems, provided the communication is of a regular nature and occurs collectively. The Crystal Router package developed at Caltech is one such example [8]. We are currently investigating how to recognize the opportunity for using Crystal Router calls that can optimally realize a collective communication sequence.

We believe that our approach can be extended to derive partitioning schemes automatically. Data dependence and other information can be used to compute a fairly restricted set of reasonable data partitioning schemes for a selected program segment. The performance estimation module can then be applied in turn to each of the partitionings in the computed set.

The possibility of performing performance estimations across procedure calls and analysis to estimate storage requirements are currently being investigated.

This tool is being implemented as part of the ParaScope parallel programming environment under development at Rice University [3].

## References

[1] J.R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing, special issue on Software Tools for Parallel Programming and Visualization*, June 1990.

[3] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. *Supercomputing 89, Reno, Nevada*, November 1989.

[4] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, October 1988.

[5] M. Chen, J. Li, and Y. Choo. Compiling parallel programs by optimizing performance. *Journal of Supercomputing*, 2:171–207, 1988.

[6] C.Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partioning for parallel computa-

tion. *International Journal of Parallel Computing*, 16, 1987.

[7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, Vol. 1.* Prentice Hall, Englewood Cliffs, NJ 07632, 1988.

[8] W. Furmanski and G. C. Fox. Optimal communication algorithms for regular decompositions on the hypercube. Technical Report C3P 314B, Caltech, March 1988.

[9] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, Bonn University, 1989.

[10] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9(4), July 1988.

[11] K. Knobe, J. Lukas, and G. Steele. Data optimization: allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[12] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting shared data structures on distributed memory machines. *Principles and Practice of Parallel Programming (PPoPP), Seattle, Washington*, March 1990.

[13] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Yale University, November 1989.

[14] J. Ramanujan and P. Sadayappan. A methodology for parallelizing programs for complex memory multiprocessors. *Supercomputing 89, Reno, Nevada*, November 1989.

[15] A. Rogers and K. Pingali. Process decomposition through locality of reference. *SIGPLAN 89 Conference on Programming Language Design and Implementation*, June 1989.

[16] M. Rosing, R. B. Schnabel, and R. P. Weaver. DINO: Summary and examples. *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988.

[17] J. H. Saltz, V. K. Naik, and D. M. Nicol. Reduction of the effects of the communication delays in scientific algorithms on message passing MIMD architectures. *SIAM Journal of Sci. and Stat. Computing*, 8(1):s118–s134, January 1987.

[18] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Boston, 1989.

[19] H.P. Zima, H-J Bast, and M Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.